In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import umap
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import RFE
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import GradientBoostingRegressor
```

Loading and preprocessing data

In [2]:
```
data_frame = pd.read_csv('Electricity BILL.csv')
data_frame
```

Out[2]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
0	Residential	1989	12	50.00000
1	Institutional	1980	6	225.75910
2	Industrial	2006	10	98.75592
3	Commercial	1985	1	68.47069
4	Industrial	2006	12	50.00000
...	...	...	...	.
1245	Residential	1985	10	147.61331
1246	Commercial	2007	5	50.00000
1247	Commercial	1990	1	50.00000
1248	Institutional	2021	6	250.00000
1249	Residential	2017	8	143.82115

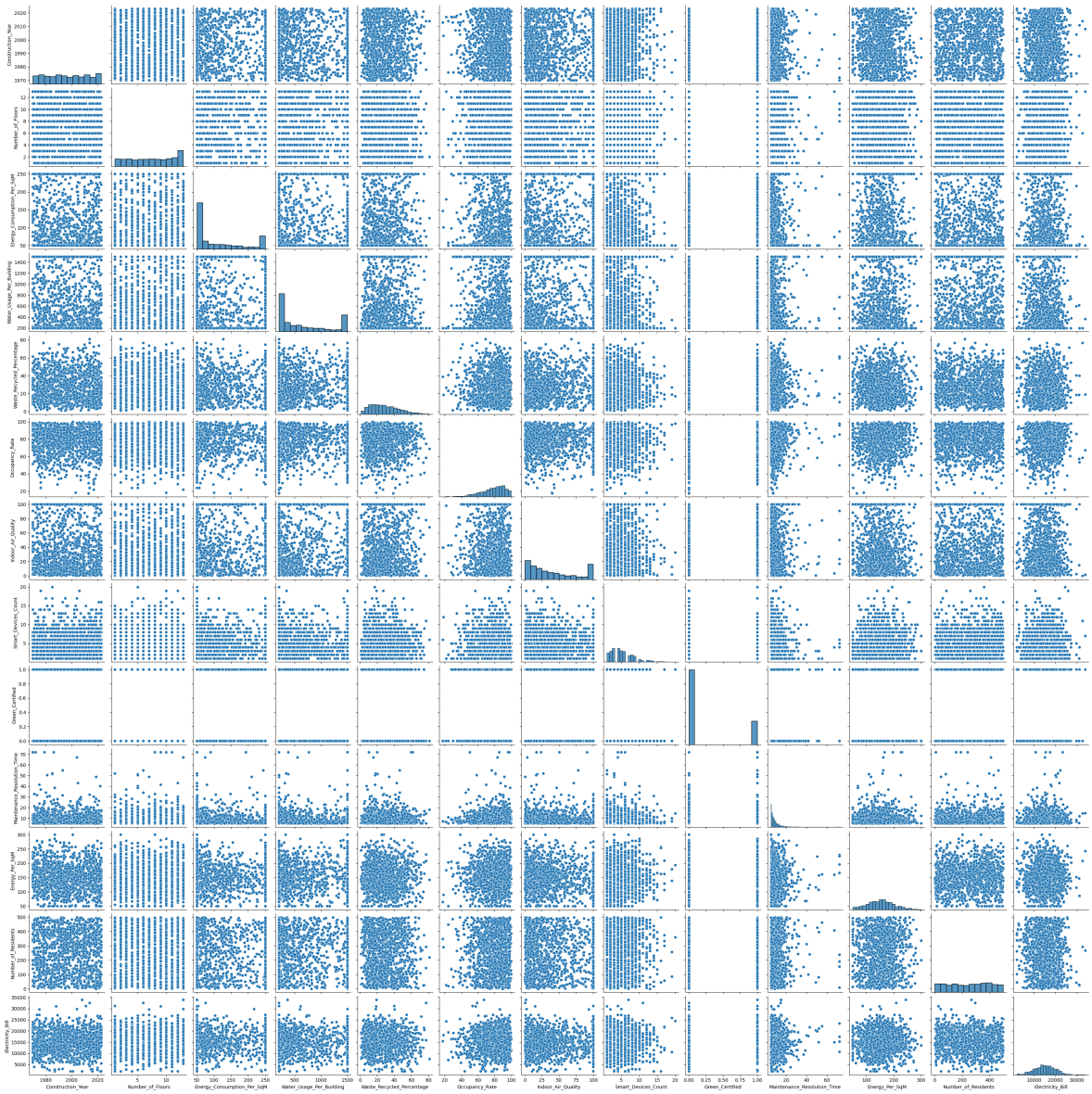
1250 rows × 16 columns

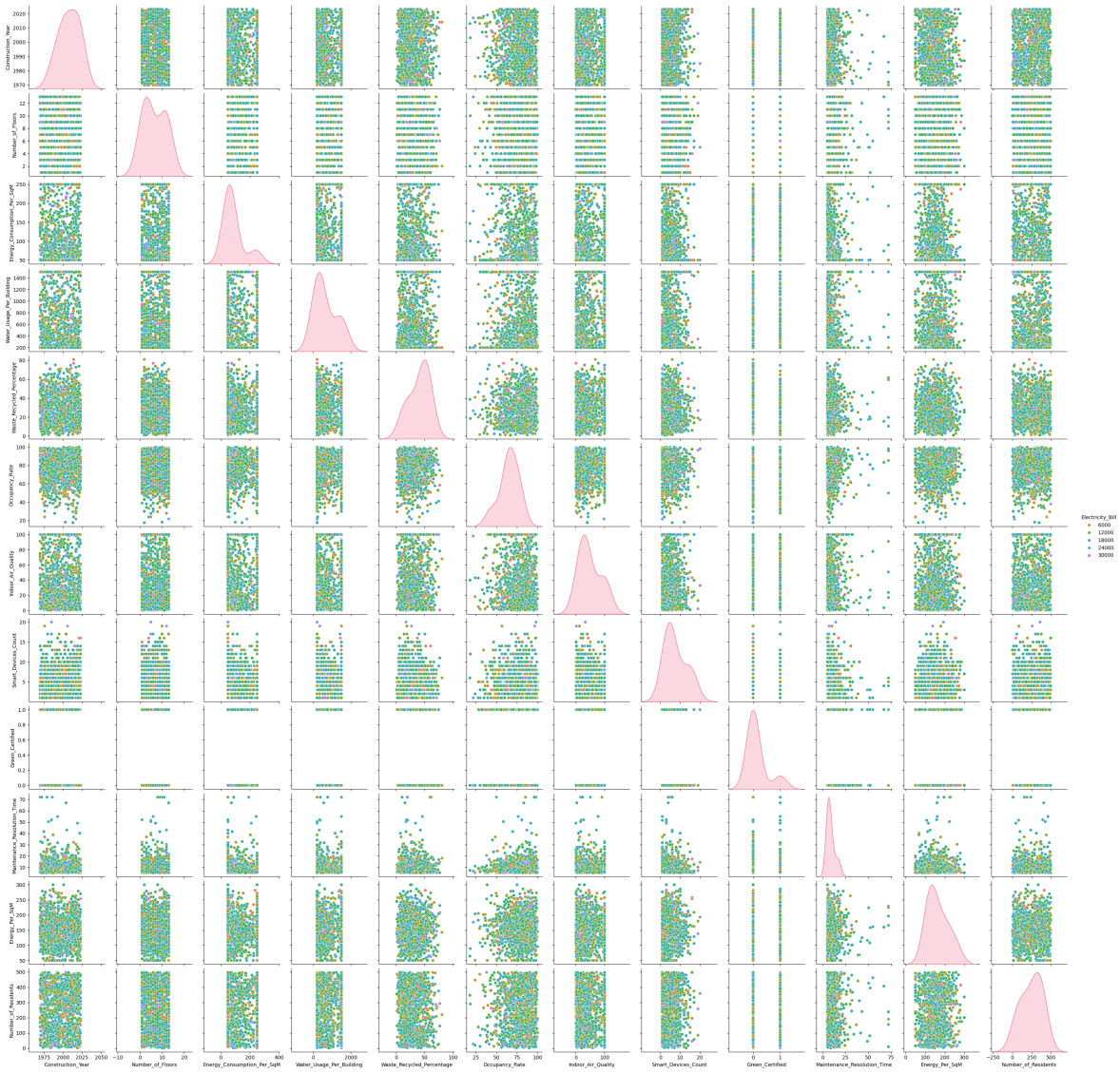
(a) EDA

PairPlot

- Pairplot is a great way to visualize the relationship between all the features in the dataset.
- Diagonal plots are univariate plots and the rest are bivariate plots.

In [3]:
```
# Perform EDA by creating pair plots, box plots, violin plots, count plots for c
# Pair plot
# plt.figsize=(20,10)
sns.pairplot(data_frame)
sns.pairplot(data_frame, hue = 'Electricity_Bill', palette = 'husl')
plt.show()
```





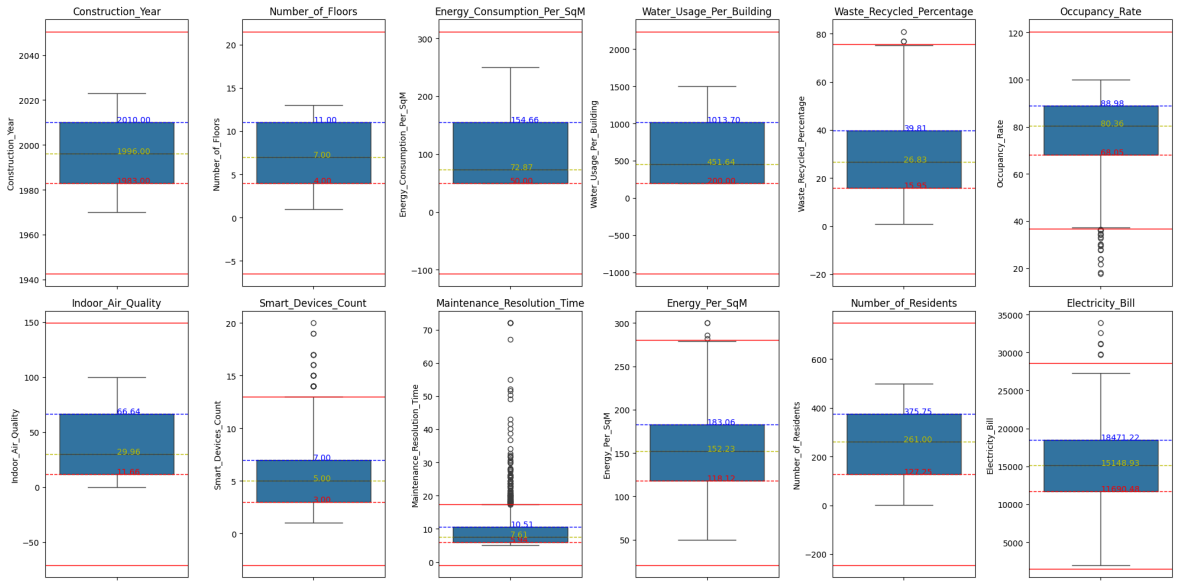
In [4]:
```
numeric_df = data_frame.select_dtypes(include=[np.number])
categorical_df = data_frame.select_dtypes(include=[object])
numeric_df1 = numeric_df.drop('Green_Certified', axis=1)
data_frame_tmp = data_frame.copy()
data_frame_tmp['Green_Certified'] = data_frame_tmp['Green_Certified'].astype(str
categorical_df1 = data_frame_tmp.select_dtypes(include=['object'])
```

Boxplot

- Boxplot is a great way to visualize the distribution of the data.
- It shows the median, quartiles, and outliers.
- It displays the following:
  - Median (Q2/50th Percentile): the middle value of the dataset.
  - First quartile (Q1/25th Percentile): the middle number between the smallest number (not the “minimum”) and the median of the dataset.
  - Third quartile (Q3/75th Percentile): the middle value between the median and the highest value (not the “maximum”) of the dataset.
  - Interquartile range (IQR): 25th to the 75th percentile.
  - Whiskers (shown in blue)
  - Outliers (shown as green points)

In [39]:
```
def calculate_boxplot_stats(data_frame, column):
    q1 = data_frame[column].quantile(0.25)
    q2 = data_frame[column].quantile(0.5)
    q3 = data_frame[column].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    return q1, q2, q3, iqr, lower_bound, upper_bound
```

In [44]:
```
# Box plot
plt.figure(figsize=(20, 10))
for i, column in enumerate(numeric_df1.columns, 1):
    plt.subplot(2, 6, i)
    sns.boxplot(y=column, data=data_frame, orient='v')

    stats = calculate_boxplot_stats(data_frame, column)

    plt.text(0, stats[0], f'{stats[0]:.2f}', color='r')
    plt.text(0, stats[1], f'{stats[1]:.2f}', color='y')
    plt.text(0, stats[2], f'{stats[2]:.2f}', color='b')


    plt.axhline(stats[0], color='r', linestyle='--', linewidth=1)
    plt.axhline(stats[1], color='y', linestyle='--', linewidth=1)
    plt.axhline(stats[2], color='b', linestyle='--', linewidth=1)
    plt.axhline(stats[4], color='r', linestyle='-', linewidth=1)
    plt.axhline(stats[5], color='r', linestyle='-', linewidth=1)

    plt.title(column)

plt.tight_layout()
plt.show()
```



Inferences from the Boxplots

1. Maintenance_Resolution_Time:

    - Very concentrated spread indicating that most of the data is closely clustered together.

- contains the highest number of outliers indicating presence of extreme cases.

2. Construction_Year and Number_of_Residents:
    - Symmetrical distribution of data with no outliers can be noticed for both cases since the whiskers are of equal length.
    - Presence of no outliers indicate that most buildings were constructed in the same year and have typically the same number of residents.
3. Wasted_Recycled_Percentage, Energy_per_SqM and Electricity_Bill:
    - Almost symmetrical distribution of data with few outliers can be noticed for all three cases.
    - This indicates that although most houses use same amount of energy, electricty and generate same amount of waste, there are few houses that are outliers and use more energy, electricity and generate more waste.
4. Energy Consumption per Sqm, Water Usage Per Building, and Indoor Air Quality:
    - Positive skewness that is the upper whisker is longer than the lower whisker and median is closer to the lower quartile.
    - This indicates that although most values lie in the lower range, there are few houses that use more energy, water and have poor indoor air quality and stretch the whisker upwards.
5. Occupancy_Rate:
    - Large number of outliers below the lower whisker indicating that a large number of houses have a low occupancy rate.

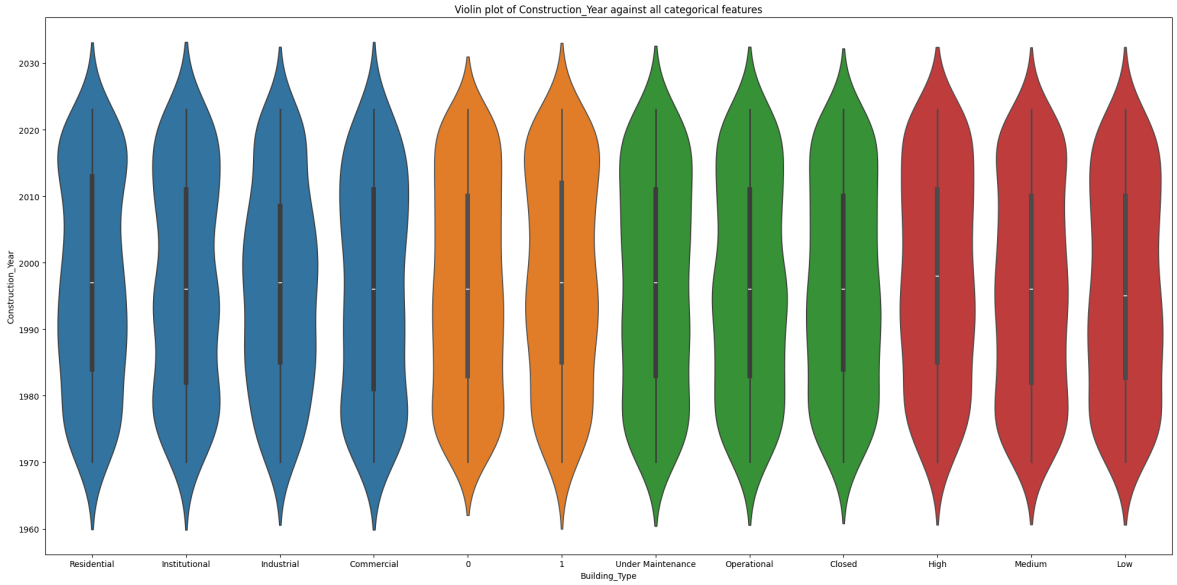
Violinplot

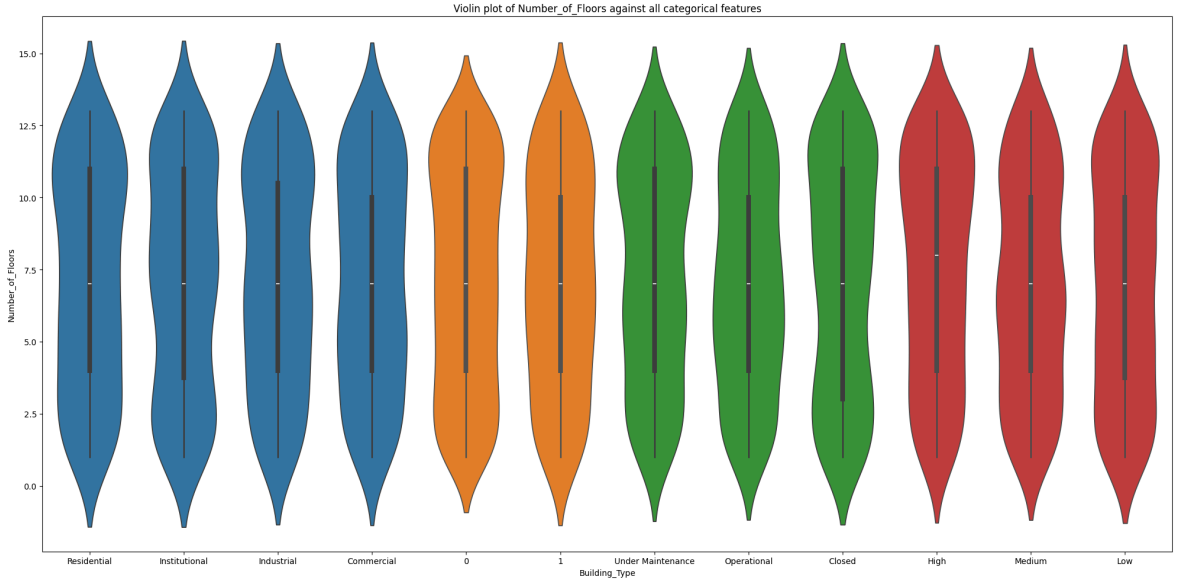
- Violinplot is a combination of a boxplot and a KDE plot.
- It shows the distribution of the data.
- It displays the following:
    - Median (Q2/50th Percentile): the middle value of the dataset.
    - First quartile (Q1/25th Percentile): the middle number between the smallest number (not the “minimum”) and the median of the dataset.
    - Third quartile (Q3/75th Percentile): the middle value between the median and the highest value (not the “maximum”) of the dataset.
    - Interquartile range (IQR): 25th to the 75th percentile.
    - Whiskers (shown in blue)
    - Outliers (shown as green points)
    - KDE plot (shown in orange)
- White centered dot: median
- Thick black line: interquartile range
- Thin black line: 95% confidence interval

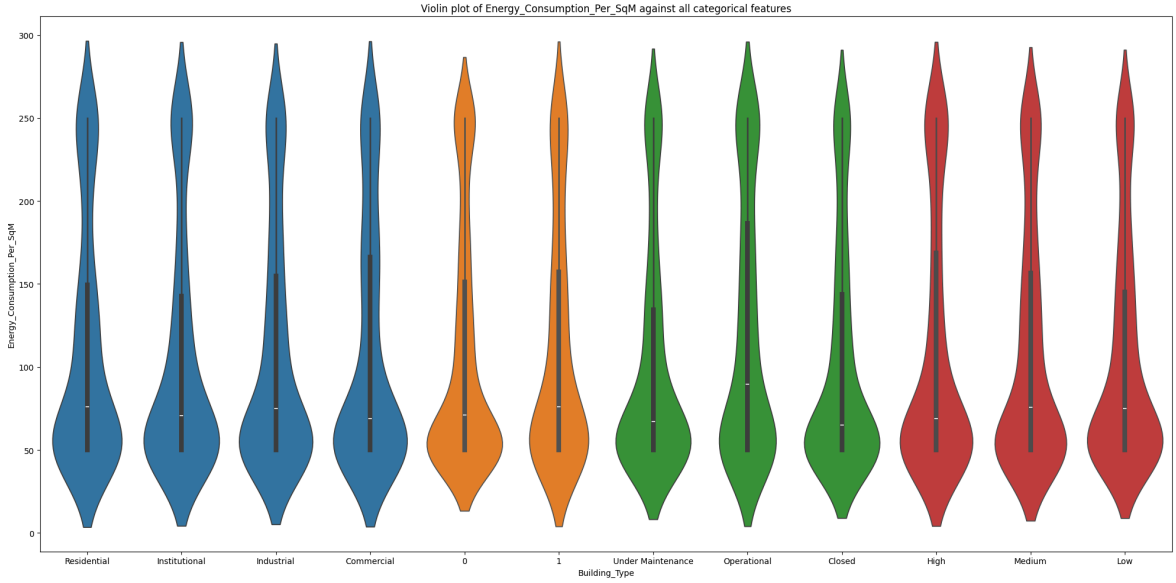
- Line boundary: distribution of the data

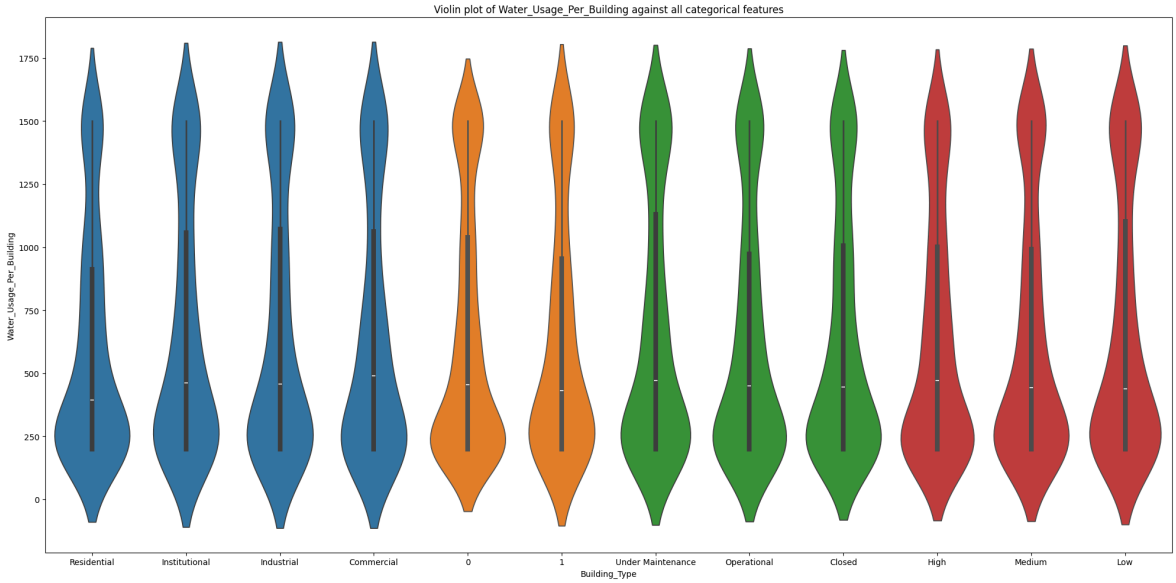
In [6]:
```
# Violin plot
categorical_col = 'Electricity_Bill'

plt.figure(figsize=(20, 10))

for num_column in numeric_df1.columns:
    plt.figure(figsize=(20, 10))
    for cat_column in categorical_df1.columns:
        sns.violinplot(x=cat_column, y=num_column, data=data_frame, orient='v')
    plt.title(f'Violin plot of {num_column} against all categorical features')
    plt.tight_layout()
    plt.show()
```

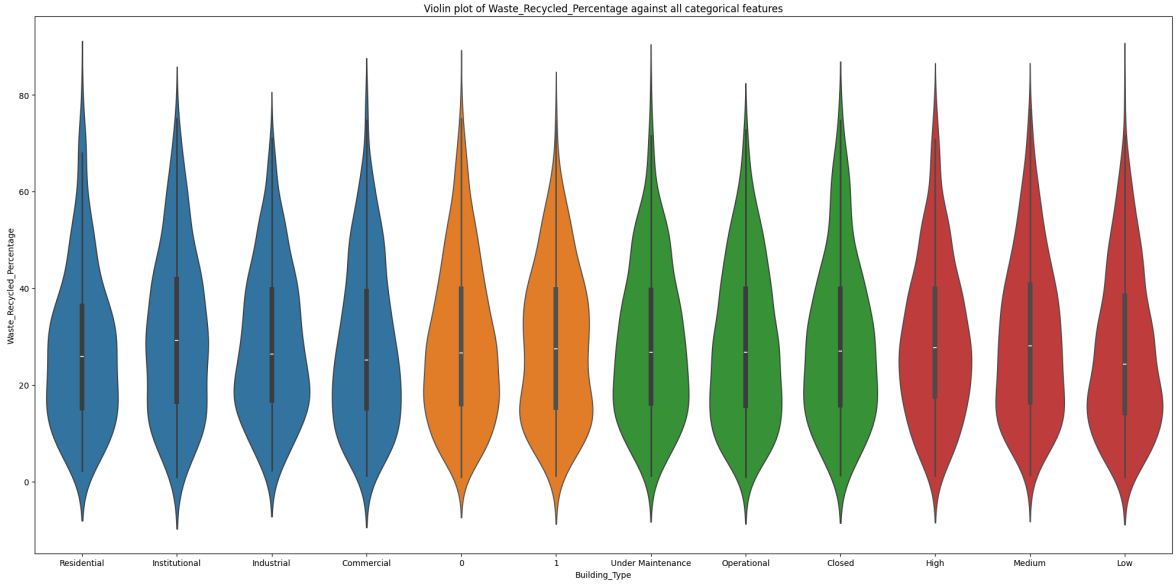
```
<Figure size 2000x1000 with 0 Axes>
```

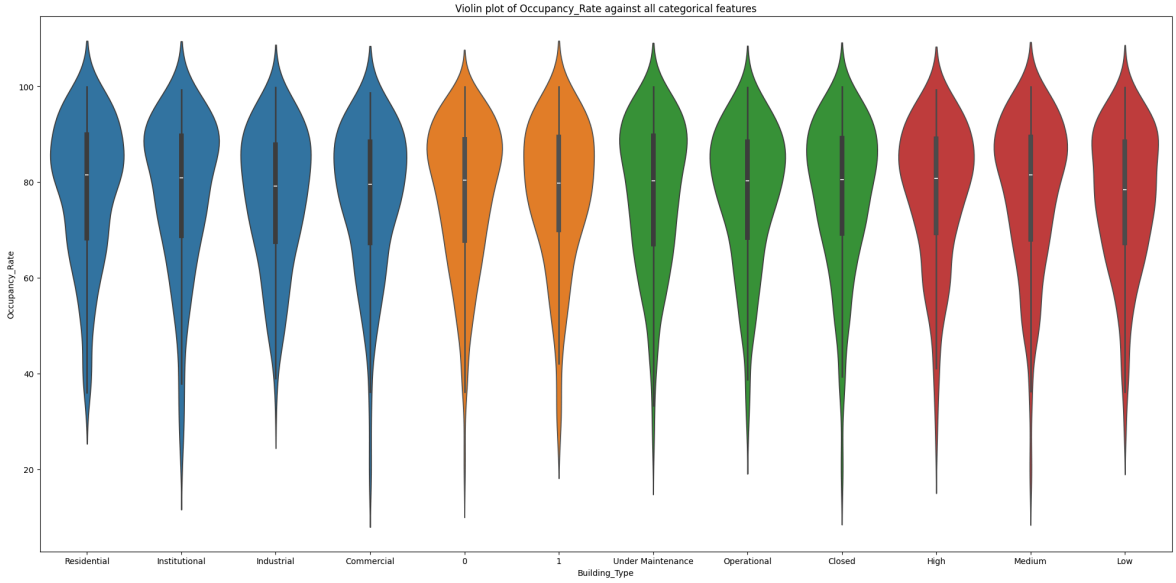


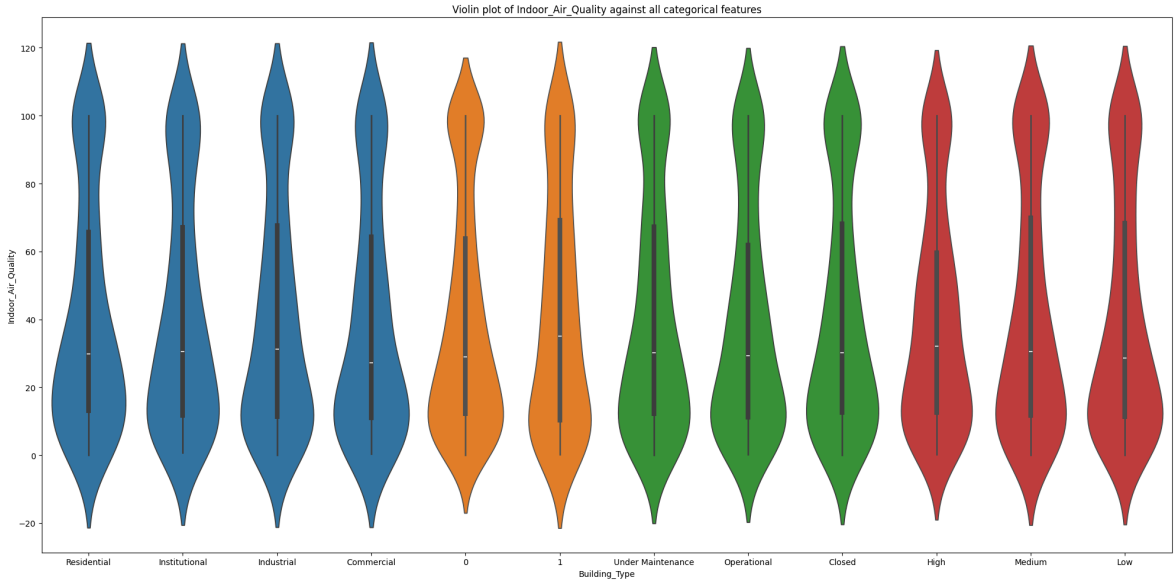


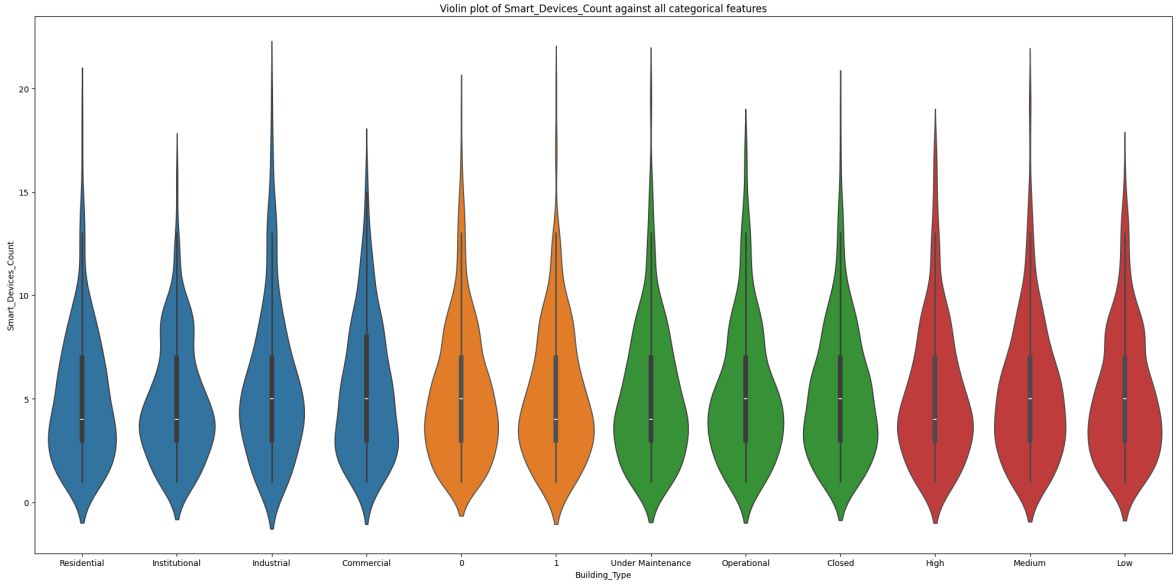

Violin plot of Energy_Consumption_Per_SqM against all categorical features
Energy_Consumption_Per_SqM
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

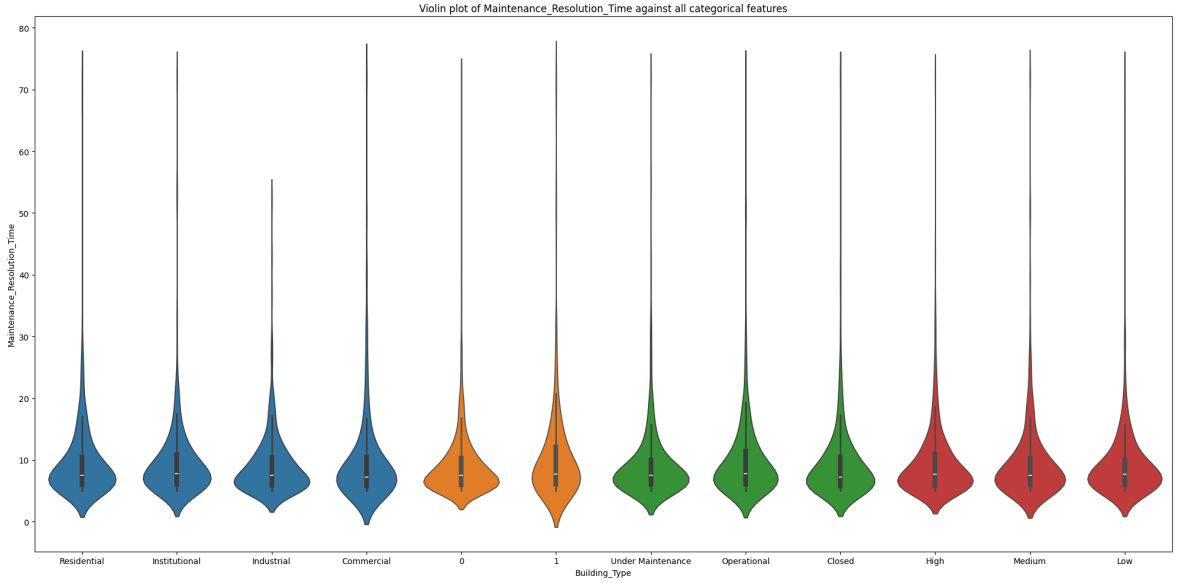

Violin plot of Water_Usage_Per_Building against all categorical features
Water_Usage_Per_Building
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

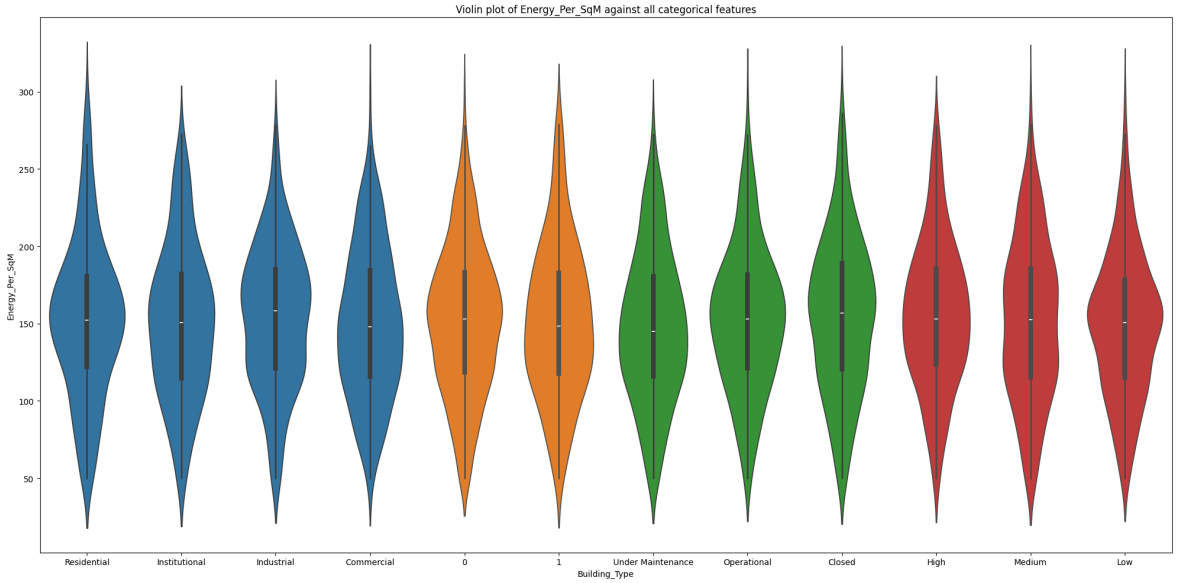

Violin plot of Waste_Recycled_Percentage against all categorical features
Waste_Recycled_Percentage
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

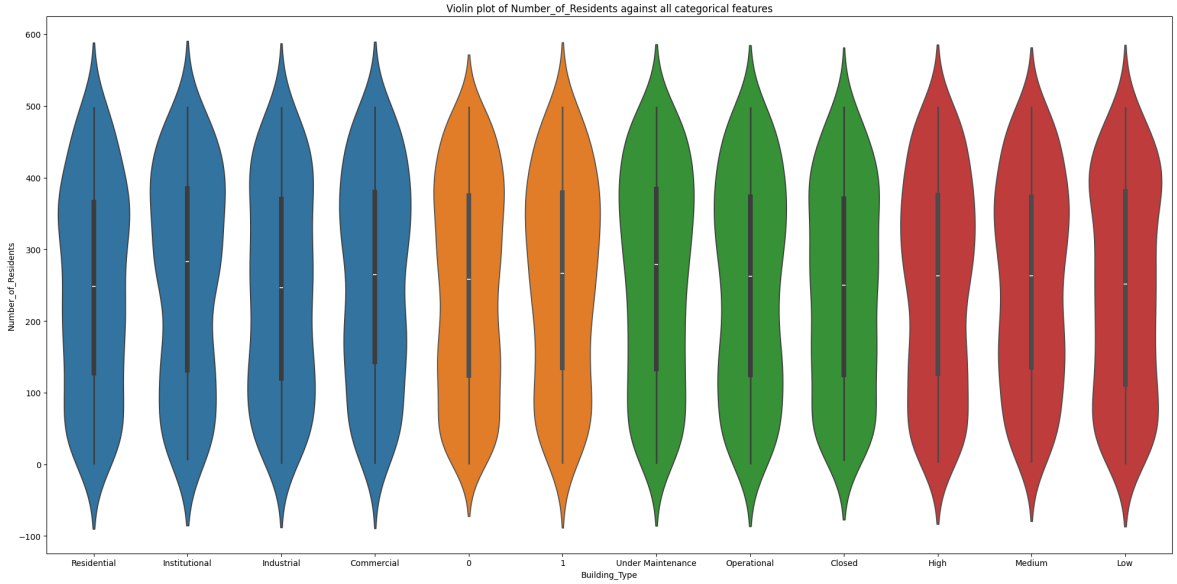

Violin plot of Occupancy_Rate against all categorical features
Occupancy_Rate
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type

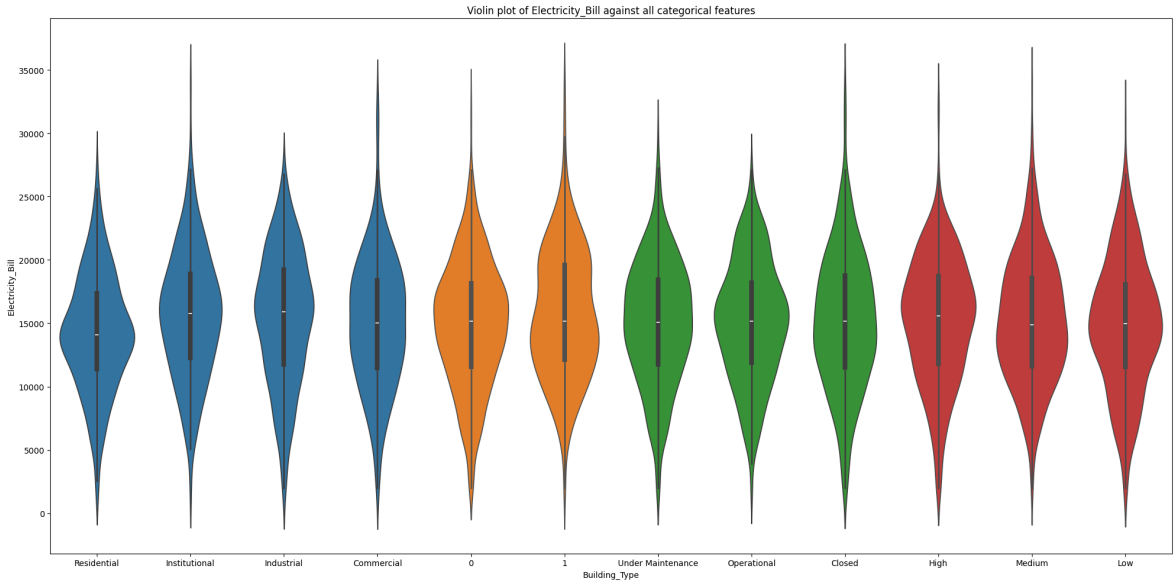

Violin plot of Indoor_Air_Quality against all categorical features
Indoor_Air_Quality
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Smart_Devices_Count against all categorical features
Smart_Devices_Count
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Maintenance_Resolution_Time against all categorical features
Maintenance_Resolution_Time
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Energy_Per_SqM against all categorical features
Energy_Per_SqM
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Number_of_Residents against all categorical features
Number_of_Residents
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type



Inferences from the Violinplot of Electricity_Bill against all the categorical features

This violin plot shows the distribution of the electricity bills across different categorical features. The wider sections of the violin plot indicate a higher density of data points, while the narrower sections represent lower density. Here's an inference for each categorical feature based on the plot:

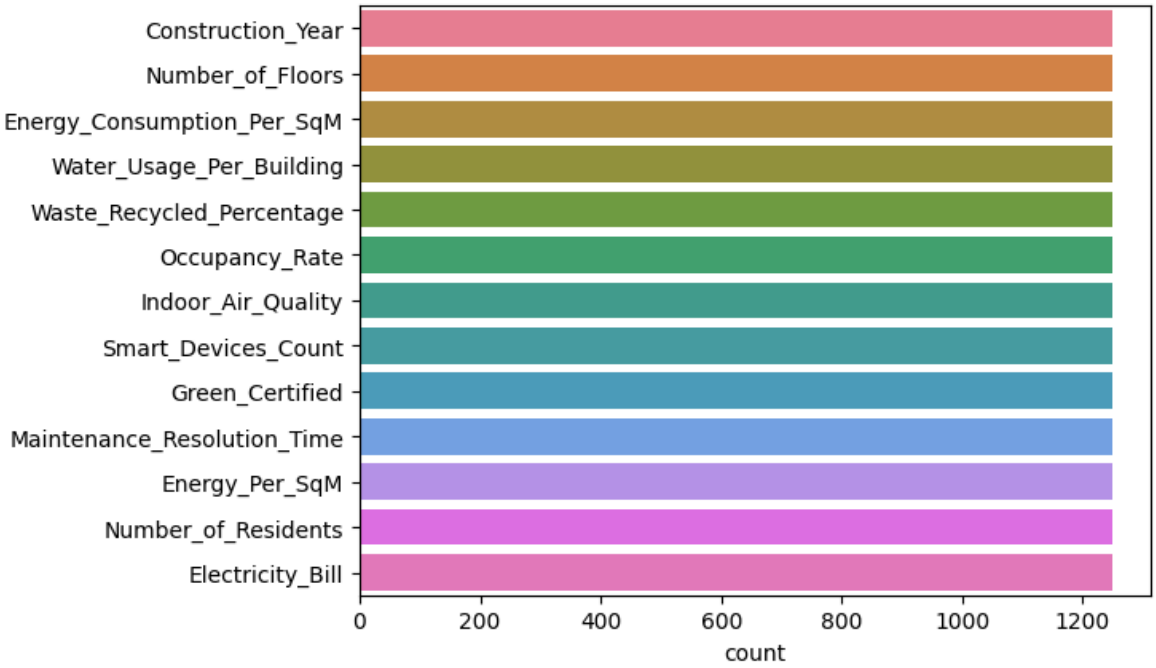
1. Building Type (Residential, Institutional, Industrial, Commercial):
    - Residential and Institutional buildings have a wide spread of electricity bills, with many bills clustering around the 15,000–20,000 range.
    - Industrial and Commercial buildings show a similar spread but appear to have higher upper limits, with some bills reaching over 30,000, though the median values remain consistent across categories. This suggests that industrial and commercial buildings may have more variability in their electricity usage compared to residential and institutional buildings.
2. Green certified (0 and 1):
    - The categories 0 and 1 likely correspond to building being green_certified or not.
    - Both categories show relatively high density around 15,000–20,000, with category 1 possibly having a slightly higher range. This suggests that both categories result in similar electricity usage patterns.
3. Building Condition (Under Maintenance, Operational, Closed):
    - Buildings Under Maintenance have a lower spread of electricity bills, likely due to lower energy usage during maintenance periods.
    - Operational buildings show a higher spread, with bills ranging from near 0 to over 35,000, indicating more active energy usage.
    - Closed buildings have a smaller spread, with most bills clustering around 15,000, but a few outliers suggesting some energy usage even when closed.
4. Energy Efficiency (High, Medium, Low):

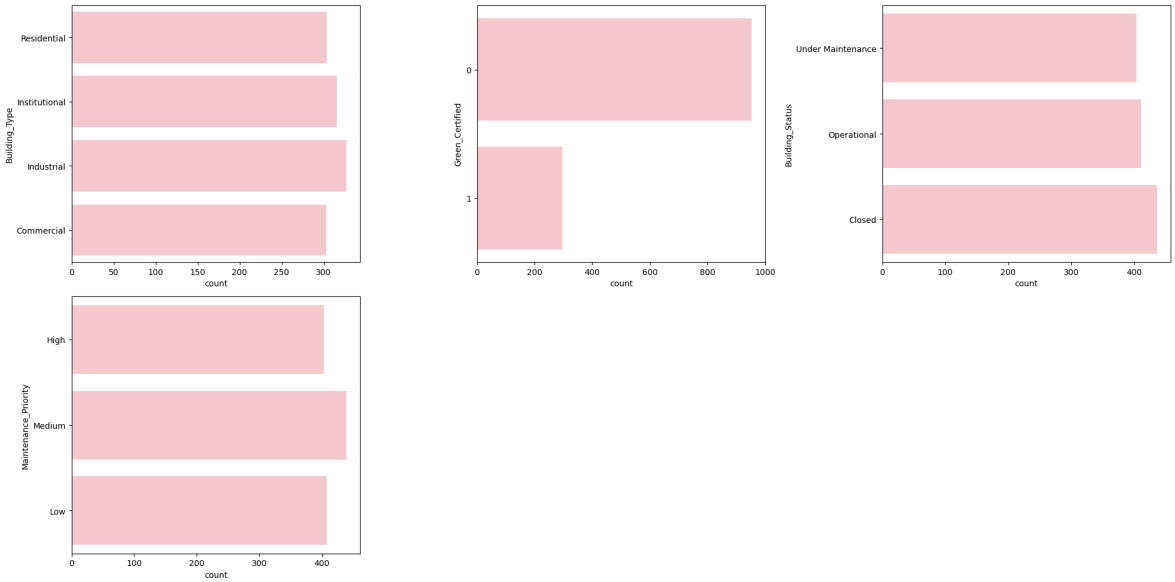
- High-efficiency buildings have a narrower spread, clustering mostly between 10,000 and 20,000, indicating more consistent and likely lower electricity usage.
- Medium and Low-efficiency buildings have a wider spread, especially for Low-efficiency buildings, where there is a greater chance of higher electricity bills, suggesting that lower efficiency correlates with higher energy consumption.

Countplot

- Countplot is a great way to visualize the count of each category in a categorical feature.
- It displays the count of each category in a bar plot.

In [7]:
```
# Count plot
sns.countplot(data_frame, orient='h')

plt.figure(figsize=(20, 10))

for i, column in enumerate(categorical_df1.columns):
    plt.subplot(2, 3, i+1)
    sns.countplot(y=column, data=data_frame_tmp, color= 'pink')

plt.tight_layout()
plt.show()
```





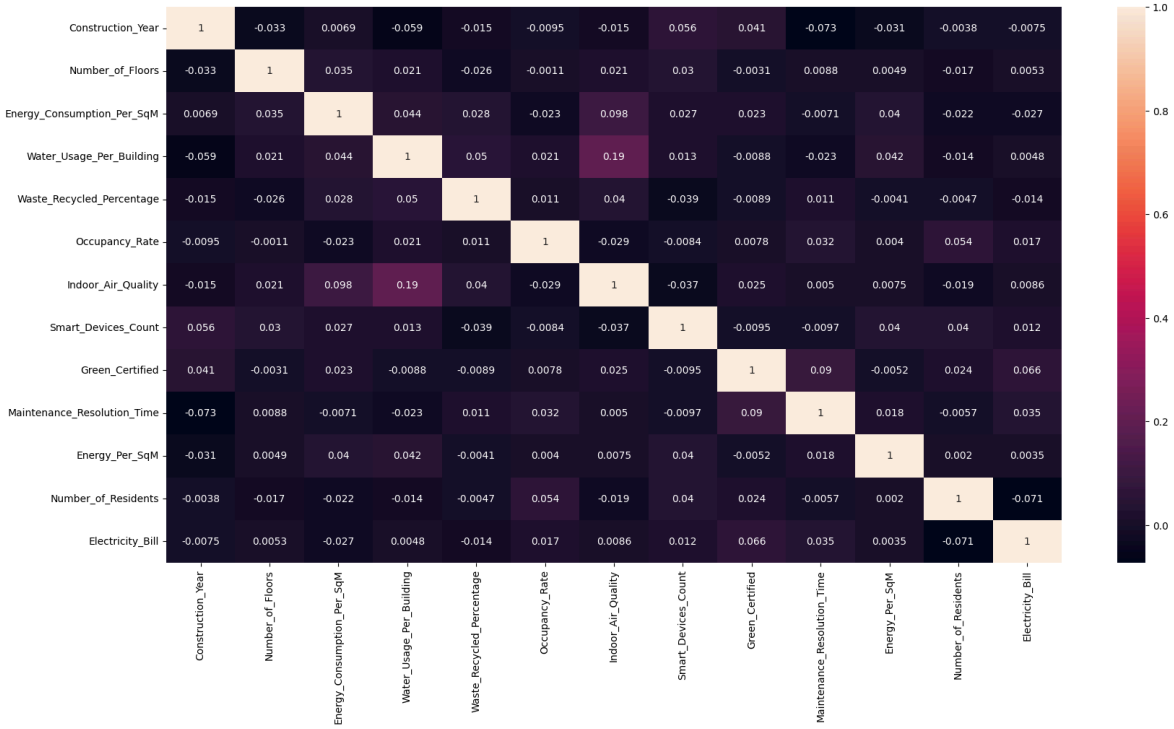
Inferences from the Countplot of the categorical features

1. Only a small fraction of buildings are Green Certified even though such houses are more sustainable.
2. Most buildings are either closed or under maintenance.

Correlation HeatMap

- Correlation heatmap is used to visualize the correlation between all the features in the dataset with colour-coding.
- The correlation coefficient ranges from -1 to 1.
- The closer the correlation coefficient is to 1, the stronger the positive correlation.
- The closer the correlation coefficient is to -1, the stronger the negative correlation.
- The closer the correlation coefficient is to 0, the weaker the correlation.
- The diagonal of the heatmap is always 1 because it represents the correlation of a feature with itself.

In [8]:
```
# Correlation heatmap

plt.figure(figsize=(20, 10))
sns.heatmap(numeric_df.corr(), annot=True)
plt.show()
```



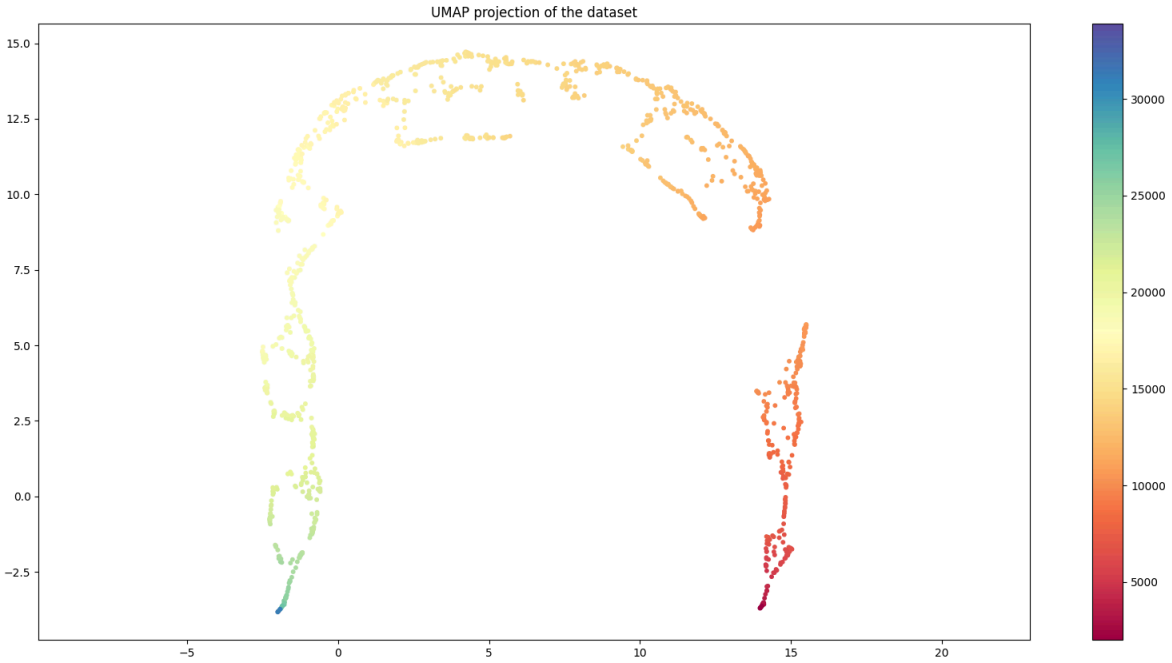
Inferences from the Correlation HeatMap

1. Energy_Consumption_per_SqM and Water_Usage_per_Building both have a strong positive correlation with Indoor_Air_Quality.
2. Maintenance_Resolution_Time and Electricity_Bill both have a strong positive correlation with Green_certified buildings.
3. Energy_Consumption_per_SqM has a strong negative correlation with Occupancy_Rate.
4. Constructio_year and Water_Usage_per_Building have a strong negative correlation.
5. Number_of_Residents and Electricity_Bill have a strong negative correlation.

(b) UMAP

UMAP is a dimensionality reduction technique that is used to visualize high-dimensional data in a lower-dimensional space. It is similar to t-SNE but is faster and scales better to large datasets. UMAP is used to visualize the data in 2D space.

In [9]:
```
# Use the Uniform Manifold Approximation and Projection (UMAP) algorithm to redu
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(numeric_df)
plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = numeric_df['Electricity_Bill']
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with Electricity_Bill labels')

plt.show()
```



Inferences from the UMAP plot of unscaled data

- We see a large number of small clusters in the UMAP plot.
- The data points show that they are not linearly separable, hence the following linear regression methods will show poor performance on this dataset.
- Some clusters are dense and some are sparse, indicating that the data is not uniformly distributed.

(c) Preprocessing data and applying Linear Regression

Min-Max Scaling for numerical features and label encoding for categorical features

In [10]:
```
# Perform the necessary pre-processing steps, including handling missing values

# data_frame.dropna(inplace=True)

scaler = MinMaxScaler()
encoder = LabelEncoder()

X = data_frame.drop(columns=['Electricity_Bill'])
y = data_frame['Electricity_Bill']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

X_train_num = X_train.select_dtypes(include=[np.number])
X_train_cat = X_train.drop(columns=X_train_num.columns)

X_test_num = X_test.select_dtypes(include=[np.number])
X_test_cat = X_test.drop(columns=X_test_num.columns)
```

```
X_train_scaled_num = pd.DataFrame(scaler.fit_transform(X_train_num), columns=X_t

# X_train_scaled_cat = X_train_cat.apply(lambda col: encoder.fit_transform(col.a

X_cat_combined = pd.concat([X_train_cat, X_test_cat])

X_cat_combined_encoded = X_cat_combined.apply(lambda col: encoder.fit_transform(

X_train_scaled_cat = X_cat_combined_encoded.loc[X_train_cat.index]
X_test_scaled_cat = X_cat_combined_encoded.loc[X_test_cat.index]


X_train_scaled_num.reset_index(drop=True, inplace=True)
X_train_scaled_cat.reset_index(drop=True, inplace=True)


X_train_scaled = pd.concat([X_train_scaled_num, X_train_scaled_cat], axis=1)


# print(X_test_num)
# print(X_test_cat)

X_test_scaled_num = pd.DataFrame(scaler.transform(X_test_num), columns=X_test_nu

# X_test_scaled_cat = X_test_cat.apply(lambda col: encoder.transform(col.astype(

X_test_scaled_num.reset_index(drop=True, inplace=True)
X_test_scaled_cat.reset_index(drop=True, inplace=True)

X_test_scaled = pd.concat([X_test_scaled_num, X_test_scaled_cat], axis=1)

data_frame_scaled = pd.concat([X_train_scaled, y_train], axis=1)
```

In [11]:
```
X_train
```

Out[11]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
1194	Residential	1994	12	50.00000
911	Institutional	1984	13	50.00000
422	Institutional	2005	4	50.00000
670	Residential	1974	1	50.00000
931	Commercial	1984	11	250.00000
...	...	...	...	.
1044	Commercial	2017	13	250.00000
1095	Commercial	2013	6	50.00000
1130	Commercial	1986	11	192.78833
860	Industrial	2015	2	224.10038
1126	Commercial	2011	3	50.00000

1000 rows × 15 columns

In [12]:
```
X_train_scaled_num
```

Out[12]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 12 columns

In [13]:
```
X_train_scaled_cat
```

Out[13]:

	Building_Type	Building_Status	Maintenance_Priority
0	3	1	1
1	2	1	1
2	2	2	1
3	3	2	2
4	0	2	0
...	...	...	...
995	0	1	2
996	0	1	1
997	0	1	0
998	1	0	0
999	0	1	2

1000 rows × 3 columns

In [14]:
```
X_train_scaled
```

Out[14]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 15 columns

In [15]:
```
y_train
```

Out[15]:
```
1194    17591.376140
911     13847.735490
422     15398.443910
670     13197.166340
931     16815.610080
           ...
1044    14828.987040
1095     5191.145924
1130    23362.030160
860      9398.009440
1126    10990.719530
Name: Electricity_Bill, Length: 1000, dtype: float64
```

In [16]:
```
X_test
```

Out[16]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
680	Residential	1991	1	250.00000
1102	Institutional	1998	10	250.00000
394	Industrial	2007	10	50.00000
930	Industrial	1996	6	50.00000
497	Commercial	2018	8	50.00000
...	...	...	...	.
382	Industrial	1977	11	74.35493
678	Industrial	1970	9	250.00000
1002	Residential	1976	13	250.00000
361	Institutional	2006	11	77.31203
490	Institutional	2004	12	50.00000

250 rows × 15 columns

In [17]:
```
X_test_scaled
```

Out[17]:

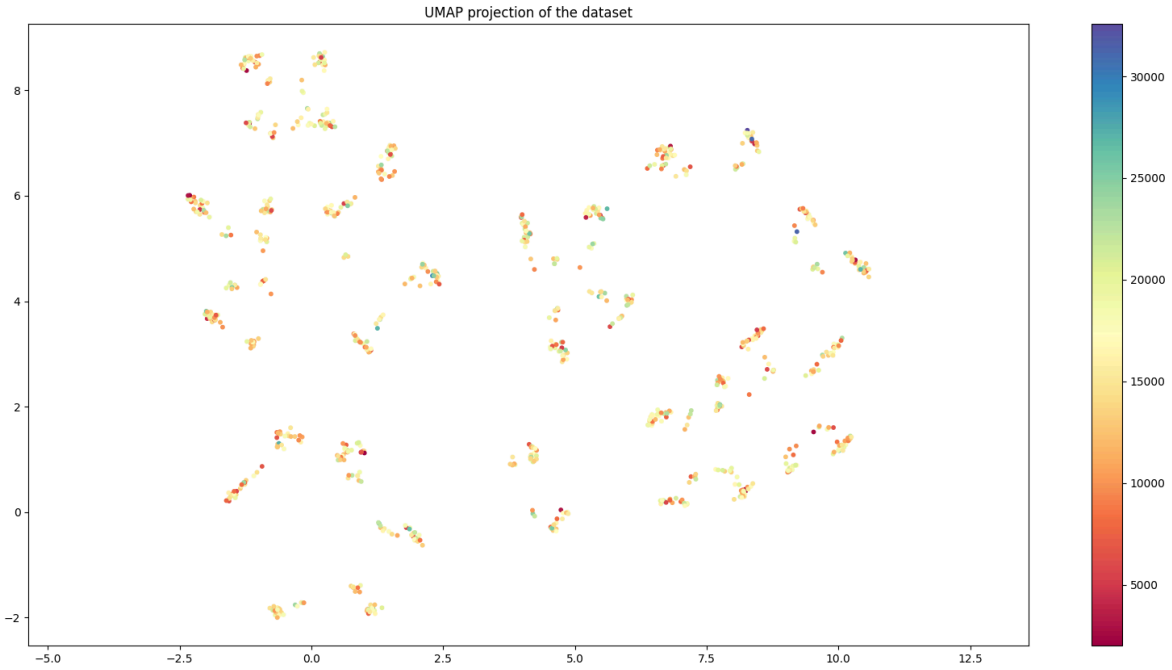
	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.396226	0.000000	1.000000	
1	0.528302	0.750000	1.000000	
2	0.698113	0.750000	0.000000	
3	0.490566	0.416667	0.000000	
4	0.905660	0.583333	0.000000	
...	...	...	...	
245	0.132075	0.833333	0.121775	
246	0.000000	0.666667	1.000000	
247	0.113208	1.000000	1.000000	
248	0.679245	0.833333	0.136560	
249	0.641509	0.916667	0.000000	

250 rows × 15 columns

In [18]:
```
y_test
```

Out[18]:
```
680      8453.948237
1102    11432.649750
394     11702.042480
930     14251.068910
497      9785.154833
            ...
382     17541.382160
678     16243.691150
1002    11987.080930
361     11816.911900
490     25433.128080
Name: Electricity_Bill, Length: 250, dtype: float64
```

In [19]:
```
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_scaled)
plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset')

plt.show()
```



Inferences from the UMAP of normalized data

- The data is divided into a large number of very small clusters.
- The data is not linearly separable however there is a more noticeable separation between the clusters.
- The data is more uniformly distributed than the unscaled data.

Linear Regression

In [20]:
```
# Applying linear regression model on pre-processed data

N = len(y_test)
p = X_test.shape[1]

N_train = len(y_train)
p_train = X_train.shape[1]

model = LinearRegression()
model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)
y_pred_train = model.predict(X_train_scaled)

R2 = model.score(X_test_scaled, y_test)
R2_train = model.score(X_train_scaled, y_train)

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train))
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_scaled, y_train))
```

```
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Mean Squared Error for train: 24475013.16847547
Mean Squared Error for test: 24278016.155742623

Root Mean Squared Error for train: 4947.222773281538
Root Mean Squared Error for test: 4927.272689403604

R2 Score for train: 0.013922520844610209
R2 Score for test: 3.7344733075372893e-05

Adjusted R2 Score for train: -0.0011091480449536562
Adjusted R2 Score for test: -0.0640628254763429

Mean Absolute Error for train: 4006.3284693293604
Mean Absolute Error for test: 3842.409312558516
```

Inferences from the Linear Regression

- The large mean squared error and root mean squared error are not only because the model is unfit for our data but also because we have not scaled the label and hence the error ends up being large.
- The R2 score of 3.7344733075372893e-05 indicates that the model is not a good fit for the data. and is performing poorly.

(d) Recursive Feature Elimination (RFE) and Correlation Analysis

RFE

In [22]:
```
# Perform Recursive Feature Elimination (RFE) or Correlation analysis on the ori

rfe = RFE(model, n_features_to_select=3)

rfe.fit(X_train_scaled, y_train)

selected_features = X_train_scaled.columns[rfe.support_]
print(f"Selected Features by RFE: {selected_features}", '\n')

X_train_rfe = X_train_scaled[selected_features]
X_test_rfe = X_test_scaled[selected_features]

model.fit(X_train_rfe, y_train)

y_pred_rfe = model.predict(X_test_rfe)
y_pred_train_rfe = model.predict(X_train_rfe)
```

```
R2_rfe = model.score(X_test_rfe, y_test)
R2_train_rfe = model.score(X_train_rfe, y_train)

N = len(y_test)
p = X_test_rfe.shape[1]

N_train = len(y_train)
p_train = X_train_rfe.shape[1]

Adjusted_R2_rfe = 1 - (1 - R2_rfe) * (N - 1) / (N - p - 1)
Adjusted_R2_train_rfe = 1 - (1 - R2_train_rfe) * (N_train - 1) / (N_train - p_tr

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train_
print('Mean Squared Error:', mean_squared_error(y_test, y_pred_rfe), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error:", np.sqrt(mean_squared_error(y_test, y_pred_rfe)

print('R2 Score for train:', model.score(X_train_rfe, y_train))
print('R2 Score:', model.score(X_test_rfe, y_test), '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train_rfe)
print("Adjusted R2 Score:", Adjusted_R2_rfe, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train_rf
print("Mean Absolute Error:", np.mean(np.abs(y_test - y_pred_rfe)), '\n')
```

```
Selected Features by RFE: Index(['Green_Certified', 'Maintenance_Resolution_Tim
e',
       'Number_of_Residents'],
      dtype='object')

Mean Squared Error for train: 24598921.04560434
Mean Squared Error: 23976300.350124482

Root Mean Squared Error for train: 4959.729936761108
Root Mean Squared Error: 4896.560052743607

R2 Score for train: 0.008930377784842514
R2 Score: 0.012464411927795571

Adjusted R2 Score for train: 0.005945228320339058
Adjusted R2 Score: 0.00042129500008580845

Mean Absolute Error for train: 4017.1253534034668
Mean Absolute Error: 3816.7223458373137
```

Inferences from the RFE

- The MSE has decreased by almost 301716 units and the R2 score has increased significantly. This indicates that RFE was successful in the selecting the three most important features for the model.

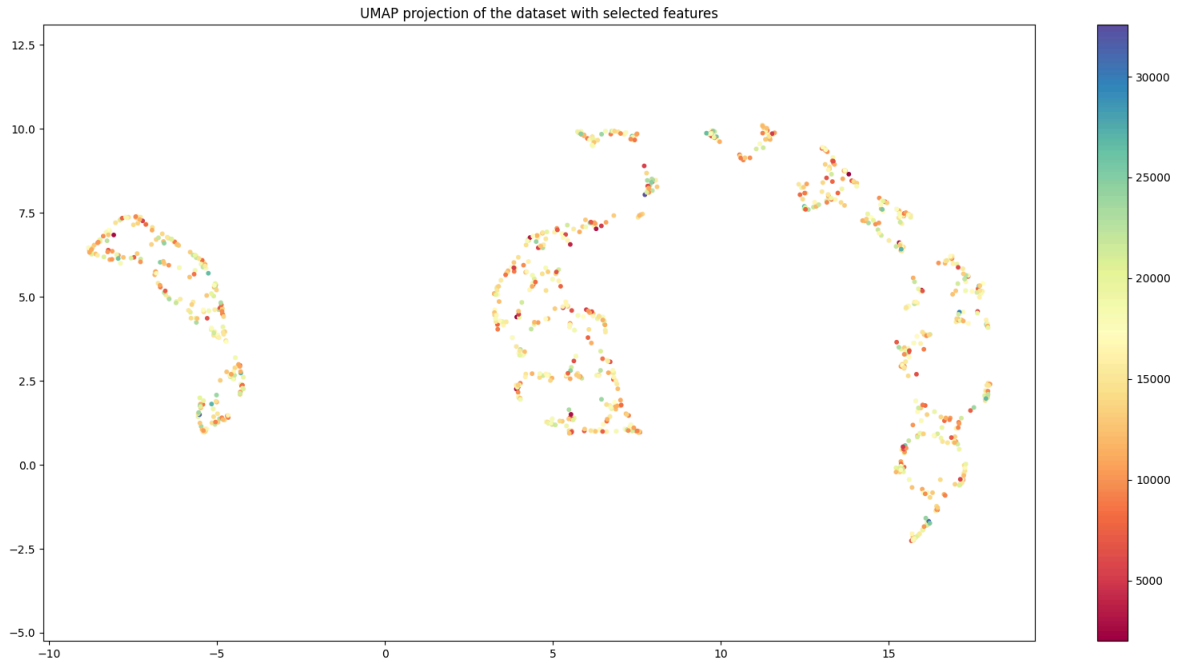
In [46]:
```
X_train_rfe
```

Out[46]:

	Green_Certified	Maintenance_Resolution_Time	Number_of_Residents
0	0.0	0.009899	0.931590
1	0.0	0.011183	0.162978
2	1.0	0.032088	0.659960
3	1.0	0.098298	0.768612
4	1.0	0.029140	0.074447
...	...	...	...
995	0.0	0.052232	0.096579
996	0.0	0.189633	0.653924
997	1.0	1.000000	0.309859
998	0.0	0.045982	0.140845
999	1.0	0.024654	0.943662

1000 rows × 3 columns

In [47]:
```
# UMAP projection of the dataset with selected features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_rfe)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with selected features')

plt.show()
```



The UMAP plot shows distribution of data in approximately three clusters due to our feature selection using RFE.

Correlation Analysis

In [24]:
```
corr_matrix = data_frame_scaled.corr()

corr_matrix['Electricity_Bill'].sort_values(ascending=False)

selected_features_corr = corr_matrix['Electricity_Bill'].sort_values(ascending=F

X_train_corr = X_train_scaled[selected_features_corr]
X_test_corr = X_test_scaled[selected_features_corr]

model.fit(X_train_corr, y_train)

y_pred_corr = model.predict(X_test_corr)
y_pred_train_corr = model.predict(X_train_corr)

R2_corr = model.score(X_test_corr, y_test)
R2_train_corr = model.score(X_train_corr, y_train)

N = len(y_test)
p = X_test_corr.shape[1]

N_train = len(y_train)
p_train = X_train_corr.shape[1]

Adjusted_R2_corr = 1 - (1 - R2_corr) * (N - 1) / (N - p - 1)
Adjusted_R2_train_corr = 1 - (1 - R2_train_corr) * (N_train - 1) / (N_train - p_

print("Features selected by correlation analysis: ", selected_features_corr, '\n

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train_
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred_corr), '

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_corr, y_train))
print('R2 Score for test:', model.score(X_test_corr, y_test), '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train_corr)
print("Adjusted R2 Score for test:", Adjusted_R2_corr, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train_co
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred_corr)), '\
```

```
Features selected by correlation analysis:  Index(['Number_of_Residents', 'Occupa
ncy_Rate', 'Building_Status'], dtype='object')

Mean Squared Error for train: 24695588.591842424
Mean Squared Error for test: 24218279.313990254

Root Mean Squared Error for train: 4969.46562437476
Root Mean Squared Error for test: 4921.207099278617

R2 Score for train: 0.005035724505008332
R2 Score for test: 0.0024977850966028114

Adjusted R2 Score for train: 0.002038844157131847
Adjusted R2 Score for test: -0.00966687606075567

Mean Absolute Error for train: 4011.4533029079444
Mean Absolute Error for test: 3823.7875391944817
```

Inferences from the Correlation Analysis

Similar to RFE, the correlation analysis method also improved the model performance, although not as significantly as RFE, indicating RFE was more effective in selecting the most important features.

In [49]:
```
X_train_corr
```

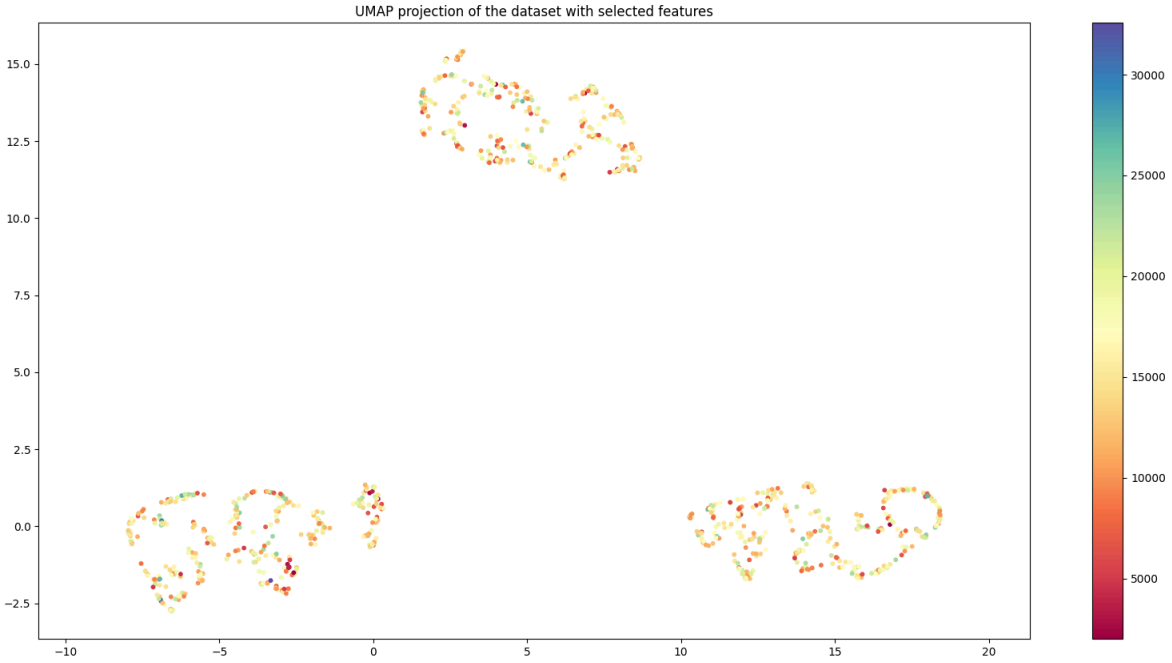
Out[49]:

	Number_of_Residents	Occupancy_Rate	Building_Status
0	0.931590	0.761137	1
1	0.162978	0.521143	1
2	0.659960	0.881262	2
3	0.768612	0.686595	2
4	0.074447	0.508822	2
...	...	...	...
995	0.096579	0.953991	1
996	0.653924	0.540006	1
997	0.309859	0.390989	1
998	0.140845	0.951936	0
999	0.943662	0.715220	1

1000 rows × 3 columns

In [48]:
```
# UMAP projection of the dataset with selected features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_corr)

plt.figure(figsize=(20, 10))
```

```
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with selected features')

plt.show()
```



The UMAP shows a huge improvement in the separability of the three clusters after feature selection using correlation analysis.

(e) One Hot Encoding and applying Ridge Regression

One Hot Encoding

In [26]:
```
# One-hot encoding categorical features
X_train_cat_encoded = pd.get_dummies(X_train_cat).reset_index(drop=True)
X_test_cat_encoded = pd.get_dummies(X_test_cat).reset_index(drop=True)

# Ensure that both train and test sets have the same columns after one-hot encod
X_train_cat_encoded, X_test_cat_encoded = X_train_cat_encoded.align(X_test_cat_e

# Concatenate the scaled numerical features with the one-hot encoded categorical
X_train_scaled_hot = pd.concat([X_train_scaled_num, X_train_cat_encoded], axis=1
X_test_scaled_hot = pd.concat([X_test_scaled_num, X_test_cat_encoded], axis=1)
```

In [28]:
```
X_train_cat_encoded
```

Out[28]:

	Building_Type_Commercial	Building_Type_Industrial	Building_Type_Institutional	Bu
0	False	False	False	
1	False	False	True	
2	False	False	True	
3	False	False	False	
4	True	False	False	
...	...	...	...	
995	True	False	False	
996	True	False	False	
997	True	False	False	
998	False	True	False	
999	True	False	False	

1000 rows × 10 columns

In [29]:
```
X_train_scaled_hot
```

Out[29]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 22 columns

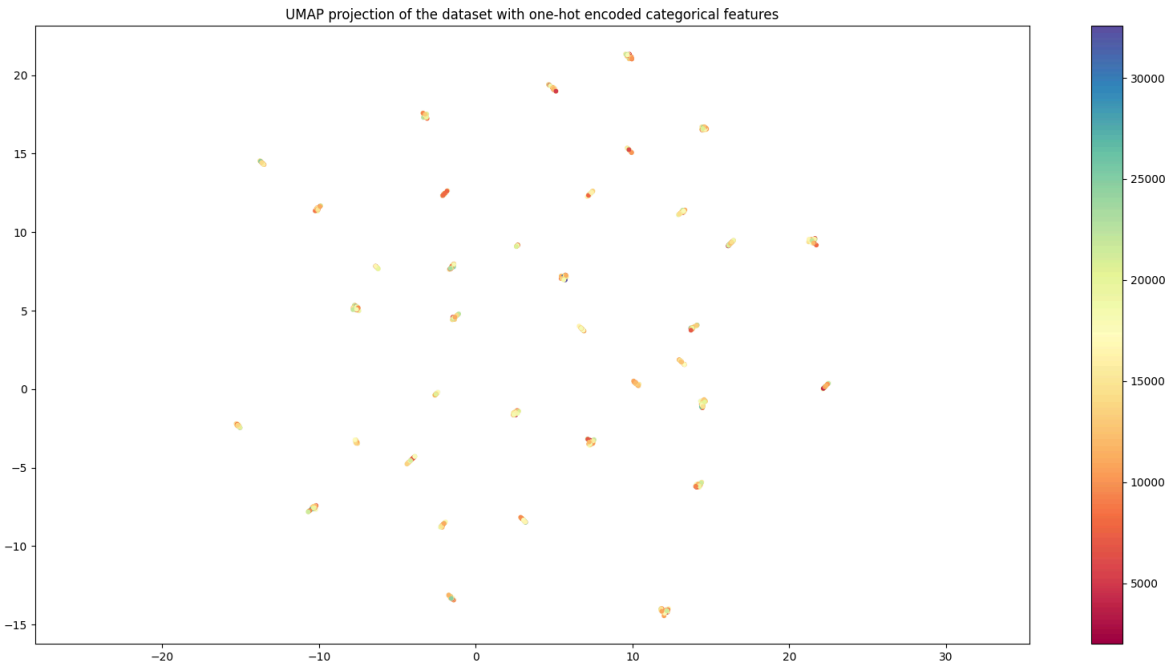
In [30]:
```
X_test_scaled_hot
```

Out[30]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.396226	0.000000	1.000000	
1	0.528302	0.750000	1.000000	
2	0.698113	0.750000	0.000000	
3	0.490566	0.416667	0.000000	
4	0.905660	0.583333	0.000000	
...	...	...	...	
245	0.132075	0.833333	0.121775	
246	0.000000	0.666667	1.000000	
247	0.113208	1.000000	1.000000	
248	0.679245	0.833333	0.136560	
249	0.641509	0.916667	0.000000	

250 rows × 22 columns

In [50]:
```
# UMAP projection of the dataset with one-hot encoded categorical features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_scaled_hot)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with one-hot encoded categorical featu

plt.show()
```



The UMAP plot shows clear distinction between various small clusters indicating that one hot encoding has improved the separability of the data.

Ridge Regression

In [31]:
```
# Applying ridge regression model on pre-processed data

model = Ridge(alpha=0.05)

model.fit(X_train_scaled_hot, y_train)

y_pred = model.predict(X_test_scaled_hot)
y_pred_train = model.predict(X_train_scaled_hot)

R2 = model.score(X_test_scaled_hot, y_test)
R2_train = model.score(X_train_scaled_hot, y_train)

N = len(y_test)
p = X_test_scaled_hot.shape[1]

N_train = len(y_train)
p_train = X_train_scaled_hot.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train))
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_scaled_hot, y_train))
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Mean Squared Error for train: 24188925.944940828
Mean Squared Error for test: 24129699.306866154

Root Mean Squared Error for train: 4918.223860799834
Root Mean Squared Error for test: 4912.199029647125

R2 Score for train: 0.0254487319342519
R2 Score for test: 0.006146217429753964

Adjusted R2 Score for train: 0.003503872264398855
Adjusted R2 Score for test: -0.09017441348013766

Mean Absolute Error for train: 3976.6864225405034
Mean Absolute Error for test: 3797.4489513870913
```

Inferences from the Ridge Regression on the One Hot Encoded data

- R2 score has seen a huge jump from 0.000037 which we obtaine dby applying linear regression on the unscaled data to 0.006.

These predictions indicate that ridge regression has done a better job at predicting the target value than logistic regression. These values are also better than regression after appyling recursive feature elimination. This implies that Ridge Regression has implemented regularization, but its effectiveness in improving model performance is limited as RMSE, MSE have improved but still the values as in are large and low R^2 score although it has a large jump. This again leads us to conclude that the model is not a good fit for the data as data is not linearly separable and needs a more complex model to predict the target value.

In [32]:
```
X_train_scaled
```

Out[32]:

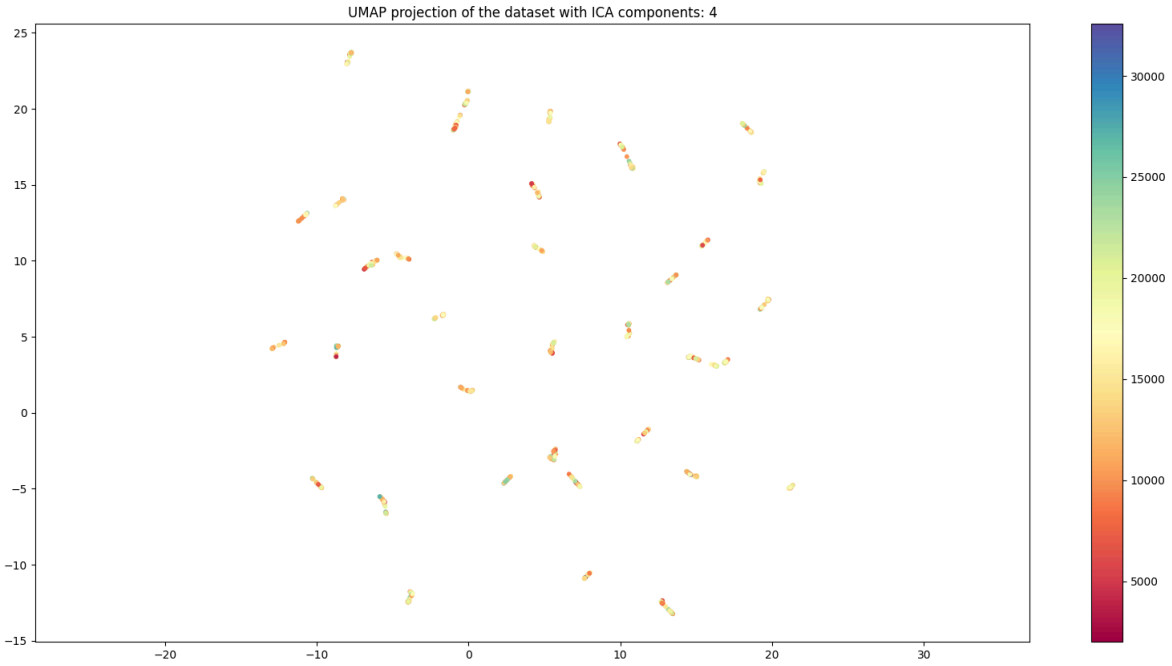
	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 15 columns

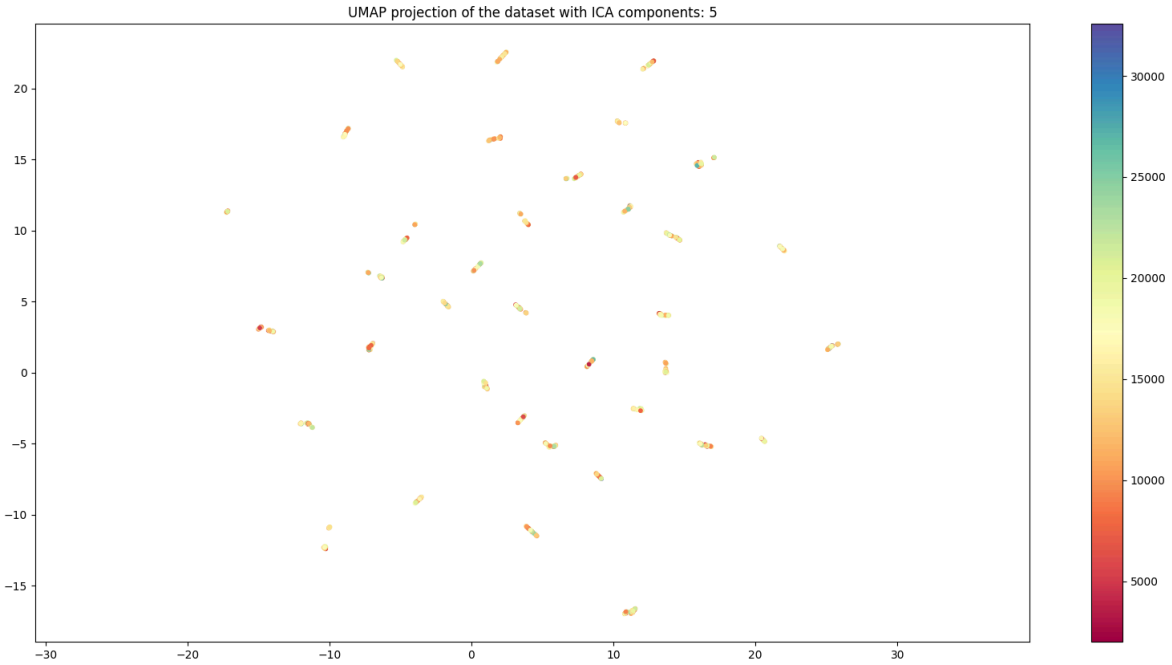
(f) Independent Component Analysis (ICA)

In [51]:
```
# Independent Component Analysis (ICA)
from sklearn.decomposition import FastICA

components = [4, 5, 6, 8]

for component in components:
    ica = FastICA(n_components=component)
    X_train_ica = ica.fit_transform(X_train_scaled_hot)
```

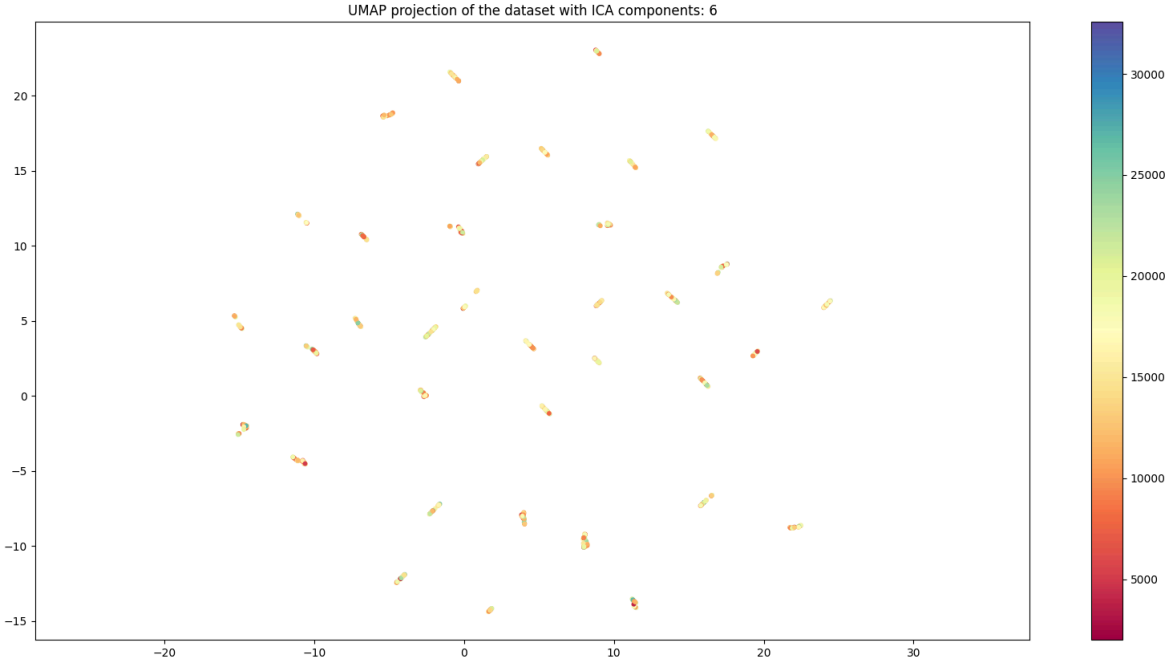
```
# UMAP projection of the dataset with ICA components
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_ica)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral'
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with ICA components: ' + str(compo

plt.show()

X_test_ica = ica.transform(X_test_scaled_hot)

model = LinearRegression()

model.fit(X_train_ica, y_train)

y_pred = model.predict(X_test_ica)
y_pred_train = model.predict(X_train_ica)

R2 = model.score(X_test_ica, y_test)
R2_train = model.score(X_train_ica, y_train)

N = len(y_test)
p = X_test_ica.shape[1]

N_train = len(y_train)
p_train = X_train_ica.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train

print("number of components: ", component, '\n')

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_tr
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_tra
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test

print('R2 Score for train:', model.score(X_train_ica, y_train))
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_trai
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n
```



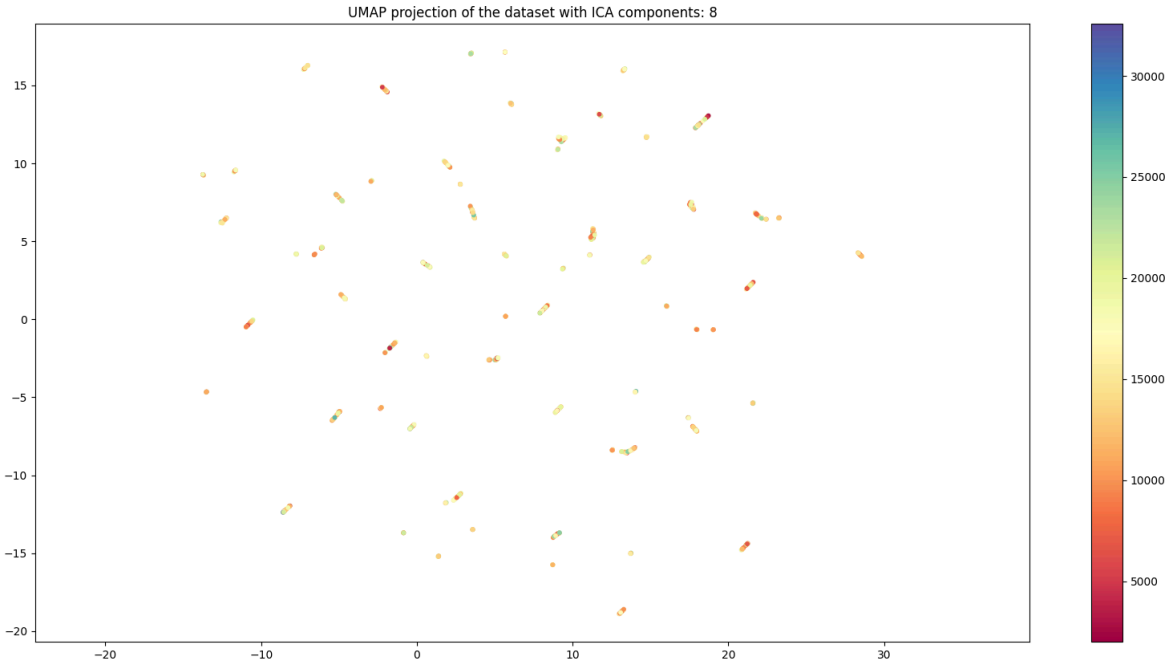

```
number of components:  4

Mean Squared Error for train: 24723510.292957045
Mean Squared Error for test: 24639889.43607532

Root Mean Squared Error for train: 4972.274157059026
Root Mean Squared Error for test: 4963.858321515162

R2 Score for train: 0.003910782897855292
R2 Score for test: -0.014867487850868777

Adjusted R2 Score for train: -9.359586436441525e-05
Adjusted R2 Score for test: -0.031436752958638126

Mean Absolute Error for train: 3999.4510999309614
Mean Absolute Error for test: 3854.3454182518394
```



```
number of components:  5

Mean Squared Error for train: 24680440.647721324
Mean Squared Error for test: 24698036.226428285

Root Mean Squared Error for train: 4967.941288674951
Root Mean Squared Error for test: 4969.711885655775

R2 Score for train: 0.005646022299354736
R2 Score for test: -0.017262437195312952

Adjusted R2 Score for train: 0.0006442417274198364
Adjusted R2 Score for test: -0.0381079789411185

Mean Absolute Error for train: 3993.727088053057
Mean Absolute Error for test: 3839.755733501596
```




```
number of components:  6

Mean Squared Error for train: 24616634.383210227
Mean Squared Error for test: 24493735.346131023

Root Mean Squared Error for train: 4961.5153313489045
Root Mean Squared Error for test: 4949.114602242609

R2 Score for train: 0.008216722467331072
R2 Score for test: -0.008847694844665854

Adjusted R2 Score for train: 0.0022240742647167133
Adjusted R2 Score for test: -0.03375751447046005

Mean Absolute Error for train: 3986.3770717768302
Mean Absolute Error for test: 3823.865663480825
```




```
number of components:  8

Mean Squared Error for train: 24414626.128019493
Mean Squared Error for test: 24074349.40562409

Root Mean Squared Error for train: 4941.115878829345
Root Mean Squared Error for test: 4906.561872189537

R2 Score for train: 0.0163554633814168
R2 Score for test: 0.008425968537081663

Adjusted R2 Score for train: 0.008414841491458547
Adjusted R2 Score for test: -0.024489352009405163

Mean Absolute Error for train: 3978.83358502663
Mean Absolute Error for test: 3781.746734661708
```

Independent Component Analysis (ICA) is a dimensionality reduction technique that is used to separate a multivariate signal into additive, independent components. It is used to separate the data into independent components.

Inferences from applying ICA

- The UMAP plot shows a clear separation between the clusters, indicating that ICA has done a good job in separating the data into independent components.
- We see that R2 score decreases from 4 to 5 components but then increases againa and is the best for 8 components on testing data but continues to increase and shows significant jump for the 8th component on the training data. MSE and RMSE similarly show a decreasing trend with increasing number of components, although not significant.

(g) ElasticNet Regularization

In [36]:
```
# elastic net regularisation

alphas = [0.01, 0.05, 0.1, 0.5, 0.9, 1]

for alpha in alphas:
    model = ElasticNet(alpha, l1_ratio=0.5, random_state=42)

    model.fit(X_train_scaled, y_train)

    y_pred = model.predict(X_test_scaled)
    y_pred_train = model.predict(X_train_scaled)

    R2 = model.score(X_test_scaled, y_test)
    R2_train = model.score(X_train_scaled, y_train)

    N = len(y_test)
    p = X_test_scaled.shape[1]

    N_train = len(y_train)
    p_train = X_train_scaled.shape[1]

    Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
    Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train

    print("alpha: ", alpha, '\n')

    print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_tr
    print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\

    print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_tra
    print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test

    print('R2 Score for train:', R2_train)
    print('R2 Score for test:', R2, '\n')

    print("Adjusted R2 Score for train:", Adjusted_R2_train)
    print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

    print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_trai
    print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n
```

```
alpha:  0.01

Mean Squared Error for train: 24478354.282398853
Mean Squared Error for test: 24248856.228290282

Root Mean Squared Error for train: 4947.560437468031
Root Mean Squared Error for test: 4924.312767106724

R2 Score for train: 0.013787910204262066
R2 Score for test: 0.0012383834956982698

Adjusted R2 Score for train: -0.0012458106767705424
Adjusted R2 Score for test: -0.06278479704944928

Mean Absolute Error for train: 4005.271158292777
Mean Absolute Error for test: 3839.381624747574

alpha:  0.05

Mean Squared Error for train: 24499905.695838828
Mean Squared Error for test: 24210445.784630973

Root Mean Squared Error for train: 4949.737942137829
Root Mean Squared Error for test: 4920.411139796244

R2 Score for train: 0.012919622073389925
R2 Score for test: 0.00282043241539387

Adjusted R2 Score for train: -0.0021273349071986303
Adjusted R2 Score for test: -0.061101334737465374

Mean Absolute Error for train: 4001.9996932398735
Mean Absolute Error for test: 3835.061865742159

alpha:  0.1

Mean Squared Error for train: 24523638.242451627
Mean Squared Error for test: 24204996.41934016

Root Mean Squared Error for train: 4952.1347157010605
Root Mean Squared Error for test: 4919.857357621271

R2 Score for train: 0.011963457940737832
R2 Score for test: 0.003044880811455708

Adjusted R2 Score for train: -0.0030980747126045927
Adjusted R2 Score for test: -0.060862498623707406

Mean Absolute Error for train: 4000.1012965943805
Mean Absolute Error for test: 3833.1716802029327

alpha:  0.5

Mean Squared Error for train: 24626260.256685033
Mean Squared Error for test: 24262212.06245762

Root Mean Squared Error for train: 4962.485290324298
Root Mean Squared Error for test: 4925.668691909518

R2 Score for train: 0.007828904206096055
```

```
R2 Score for test: 0.0006882835488326577

Adjusted R2 Score for train: -0.0072956551810061665
Adjusted R2 Score for test: -0.06337015981342153

Mean Absolute Error for train: 3997.4211264822675
Mean Absolute Error for test: 3833.3753921659923

alpha:  0.9

Mean Squared Error for train: 24668644.48692768
Mean Squared Error for test: 24291637.799691353

Root Mean Squared Error for train: 4966.753918499253
Root Mean Squared Error for test: 4928.654765723742

R2 Score for train: 0.006121279592130824
R2 Score for test: -0.0005237033840645999

Adjusted R2 Score for train: -0.009029310657989065
Adjusted R2 Score for test: -0.06465983821637633

Mean Absolute Error for train: 3998.035490384184
Mean Absolute Error for test: 3835.4456764628862

alpha:  1

Mean Squared Error for train: 24675807.87655933
Mean Squared Error for test: 24296288.079981916

Root Mean Squared Error for train: 4967.475000094045
Root Mean Squared Error for test: 4929.126502736759

R2 Score for train: 0.005832672712062448
R2 Score for test: -0.0007152390761595573

Adjusted R2 Score for train: -0.009322317033180427
Adjusted R2 Score for test: -0.06486365183745191

Mean Absolute Error for train: 3998.2727677958237
Mean Absolute Error for test: 3835.7496650742287
```

Inferences from applying ElasticNet Regularization

- As the value of alpha increases from 0.001 to 0.1 we see an increase in the R2 score but then it starts decreasing again and shows a significant drop at 1.0.
- We also see that R2 score for training data keeps on decreasing with increasing alpha values. This suggests that imposin higher penalty on the model is not a good idea as it leads to underfitting of the model.

Note: Our L1 ratio is 0.5 which means that we are using a combination of L1 and L2 regularization.

(h) Gradient Boosting Regression

In [37]:
```
# Gradient Boosting Regressor

model = GradientBoostingRegressor(n_estimators=5000, learning_rate=0.1, random_s

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)
y_pred_train = model.predict(X_train_scaled)

R2 = model.score(X_test_scaled, y_test)
R2_train = model.score(X_train_scaled, y_train)

N = len(y_test)
p = X_test_scaled.shape[1]

N_train = len(y_train)
p_train = X_train_scaled.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print("Iterations: ", 5000, "learning rate: ", 0.1, '\n')

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train)
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', R2_train)
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Iterations:  5000 learning rate:  0.1

Mean Squared Error for train: 7.248172357591522
Mean Squared Error for test: 28777559.87192035

Root Mean Squared Error for train: 2.692242997500694
Root Mean Squared Error for test: 5364.472003088501

R2 Score for train: 0.999999707977296
R2 Score for test: -0.18528981103844022

Adjusted R2 Score for train: 0.9999997035257304
Adjusted R2 Score for test: -0.261269927130648

Mean Absolute Error for train: 2.1096410858446415
Mean Absolute Error for test: 4232.535371799777
```

Inferences from applying Gradient Boosting Regression

- On training data we see hugely significant improvement in metrics as the R2 score comes close to 1 and MSE and RMSE are very low. This suggests that the model is trained very well on the training data.
- However, on testing data we see that the R2 score is very low and MSE and RMSE are very high. This suggests that the model is overfitting on the training data and is not able to generalize well on the testing data.
- As compared to ElasticNet Regularization, Gradient Boosting Regression has performed better on the training data but has not been able to generalize well on the testing data. Hence, ElasticNet Regularization is better when it comes to generalizing on unseen data but Gradient Boosting Regression is better at training on the training data.

In [ ]: